UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LIMITAÇÃO DO ESPAÇO DE ARMAZENAMENTO DE *CHECKPOINTS* EM SIMULAÇÃO DISTRIBUÍDA OTIMISTA

MAICON GHIDOLIN

CHAPECÓ
2018

MAICON GHIDOLIN

LIMITAÇÃO DO ESPAÇO DE ARMAZENAMENTO DE *CHECKPOINTS* EM SIMULAÇÃO DISTRIBUÍDA OTIMISTA

Trabalho de conclusão de curso de graduação apresentado como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Prof. Dr. Braulio Adriano de Mello

Co-orientador: Ricardo Parizotto

CHAPECÓ
2018

PROGRAD/DBIB - Divisão de Bibliotecas

Ghidolin, Maicon
LIMITAÇÃO DO ESPAÇO DE ARMAZENAMENTO DE CHECKPOINTS EM SIMULAÇÃO DISTRIBUÍDA OTIMISTA/ Maicon Ghidolin. -- 2018.
35 f.:il.

Orientador: Braulio Adriano de Mello.
Co-orientador: Ricardo Parizotto.
Trabalho de conclusão de curso (graduação) - Universidade Federal da Fronteira Sul, Curso de Ciência da Computação , Chapecó, SC, 2018.

1. Simulação . 2. Simulação Distribuída Otimista. 3. Checkpoints inúteis. 4. Limitação do espaço de armazenamento de checkpoints. I. Mello, Braulio Adriano de, orient. II. Parizotto, Ricardo, co-orient. III. Universidade Federal da Fronteira Sul. IV. Título.

Elaborada pelo sistema de Geração Automática de Ficha de Identificação da Obra pela UFFS com os dados fornecidos pelo(a) autor(a).

MAICON GHIDOLIN

LIMITAÇÃO DO ESPAÇO DE ARMAZENAMENTO DE *CHECKPOINTS* EM SIMULAÇÃO DISTRIBUÍDA OTIMISTA

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dr. Braulio Adriano de Mello

Aprovado em: 09 \ 07 \ 2018

BANCA EXAMINADORA:

Dr. Braulio Adriano de Mello - UFFS

Ricardo Parizotto - UFFS

Dr. Claunir Pavan - UFFS

Dr. Emílio Wuerges - UFFS

Let's go invent tomorrow instead of worrying about what happened yesterday
— STEVE JOBS

RESUMO

Modelos de simulação distribuída que possuem componentes assíncronos estão sujeitos a retrocesso no tempo causados por violação de tempo. *Checkpoints* são utilizados para restauração de estados consistentes anteriores à violação de tempo. Ao longo da simulação os componentes podem gerar *checkpoints* inúteis em caso de retrocesso. Em simulações grandes esses *checkpoints* inúteis ocupam cada vez mais memória, e a falta de um armazenamento limitado desses *checkpoints* pode levar a simulação a um colapso por falta de memória. O objetivo deste trabalho é apresentar algumas técnicas de limitação do espaço de armazenamento de *checkpoints* e algumas estratégias para indicar quando os *checkpoints* podem ser removidos sem prejuízo para a simulação. As técnicas e estratégias serão implementadas no DCB (*Distributed Co-simulation Backbone*) a fim de fazer experimentos para avaliar o desempenho e a utilidade dos resultados de acordo com o objetivo proposto.

Palavras-chave: Simulação distribuída, Componentes assíncronos, Checkpoints, Memória limitada.

ABSTRACT

Distributed simulation models that have asynchronous components are subject to time backtracking caused by time violation. *Checkpoints* are used for restoring consistent states prior to time violation. Throughout the simulation the components can generate non-useful *checkpoints* in case of throwback. In large simulations these useless checkpoints take up more and more memory, and the lack of limited storage of these checkpoints can lead the simulation to a collapse due to lack of memory. The objective of this work is to present some techniques of limiting the storage space of *checkpoints* and some strategies to indicate when the *checkpoints* can be removed without prejudice to the simulation. The techniques and strategies will be implemented in Distributed Co-simulation Backbone (DCB) in order to perform experiments to evaluate the performance and usefulness of the results according to the proposed objective.

Keywords: Distributed simulation, Asynchronous components, Checkpoints, Limited memory.

LISTA DE FIGURAS

Figura 2.1 – LCC - Violação de tempo 15
Figura 2.2 – Estado global consistente e inconsistente (ELNOZAHY et al., 2002) 16
Figura 2.3 – Exemplo de *checkpointing* 16
Figura 2.4 – Exemplo de *checkpoints* inúteis 18
Figura 2.5 – Arquitetura do DCB 19
Figura 4.1 – Possíveis *Recovery Lines* 23
Figura 4.2 – Modelo de troca de mensagens 28
Figura 4.3 – Tempo total de execução da simulação 30
Figura 4.4 – Total de mensagens trocadas 31
Figura 4.5 – Total de *checkpoints* 31

LISTA DE ABREVIATURAS E SIGLAS

DCB	*Distributed Co-Simulation Backbone*
DCBK	*Distributed Co-Simulation Backbone Kernel*
DCBR	*Distributed Co-Simulation Backbone Receiver*
DCBS	*Distributed Co-Simulation Backbone Sender*
GVT	*Global Virtual Time*
LAN	*Local Area Network*
LCC	*Local Causality Constraint*
LVT	*Local Virtual Time*
PL	*Processo Lógico*
WAN	*Wide Area Network*

SUMÁRIO

1 INTRODUÇÃO 11
2 SIMULAÇÃO DISTRIBUÍDA 14
2.1 Simulação 14
2.2 Simulação Distribuída 14
2.2.1 Simulação Síncrona (Conservadora) 15
2.2.2 Simulação Assíncrona (Otimista) 15
2.3 Gerenciamento de Falhas 15
2.4 *Checkpoints* 16
2.5 *Checkpoints* Inúteis 17
2.6 Eliminação de *Checkpoints* 18
2.7 DCB 18
3 TRABALHOS RELACIONADOS 20
4 DESENVOLVIMENTO 22
4.1 Contextualização da Solução 22
4.2 Algoritmo 23
4.3 Sincronização e Checkpointing no DCB 27
4.4 Implementação do algoritmo no DCB 27
4.5 Modelo 28
4.6 Estudo de caso e resultados 29
5 CONCLUSÃO 33
5.1 Trabalhos futuros 33
REFERÊNCIAS 34

1 INTRODUÇÃO

A ideia e propósito principal da simulação computacional é representar o comportamento de sistemas reais através de modelos que imitam suas caraterísticas.

A adoção de técnicas de simulação se torna uma alternativa que viabiliza experimentos em sistemas perigosos ou de alto custo, onde testes em situações reais seriam impraticáveis. Além disso, métodos de simulação auxiliam no entendimento de sistemas existentes, na análise de novos sistemas antes de sua implementação e na análise de performance de sistemas existentes em condições variadas (LAW; KELTON, 1997).

Uma grande simulação pode ser dividida em sub-simulações que podem ser executadas simultaneamente, reduzindo assim o tempo de execução (FUJIMOTO, 2000). Estas sub-simulações (ou componentes) podem ser executadas em plataformas multiprocessadas, em computadores conectados por uma rede local (LAN) ou em computadores geograficamente distribuídos, conectados via *Wide Area Network (WAN)* (FUJIMOTO, 2001).

Quando componentes são executados distribuidamente, políticas de gerenciamento de tempo se tornam necessárias, uma vez que o tempo não é mais controlado por uma única máquina. O gerenciamento de tempo é responsável por garantir a sincronização da simulação, ou seja, garantir que os eventos ocorram na ordem correta (FUJIMOTO, 2001). Eventos são ações que alteram o estado do modelo. Para isso, cada componente controla seu tempo local (LVT - *Local Virtual Time*) e o tempo global (GVT - *Global Virtual Time*) é calculado pelo maior LVT para simulações síncronas e pelo menor LVT para simulações assíncronas.

Uma simulação é dita síncrona (conservadora) quando não aceita violações de tempo, ou seja, um processo somente executa um evento quando tiver a garantia que nenhum outro evento com *timestamp* menor vai ser solicitado (FUJIMOTO, 2001). O *timestamp* é uma informação adicionada na mensagem que solicita a execução de um evento e informa ao componente de destino qual é o tempo que o evento deve ser executado.

Ao contrário de uma simulação síncrona, uma simulação é dita assíncrona (otimista) quando permite que violações de tempo (LCC) ocorram, ou seja, quando permite que um componente solicite a execução de um evento com *timestamp* menor que o LVT do componente de destino. Neste caso o componente de destino é forçado a refazer a simulação a partir de um ponto livre de erros (estado seguro) (QUAGLIA, 1999) (FUJIMOTO, 2001).

A criação de *checkpoints* é uma forma de suporte à restauração de estados seguros na

ocorrência de violações de tempo. Um processo salva periodicamente seu estado para que, em caso de falha, possa retornar para o último estado global consistente (CAO; SINGHAL, 1998). Entretanto, o processo de salvamento de *checkpoints* consome recursos de armazenamento (ELNOZAHY et al., 2002) e consequentemente o salvamento com espaço de endereçamento ilimitado desses *checkpoints* pode acarretar em uma falta de memória, principalmente em simulações grandes, podendo causar até a sua interrupção. Embora existam estudos que proponham métricas e técnicas para reduzir a criação de *checkpoints* inúteis (QUAGLIA, 1999), em geral os ambientes e arquiteturas de simulação não incorporam mecanismos que limitam o armazenamento de *checkpoints*. Um exemplo disso é o DCB, que é uma arquitetura de suporte a simulação distribuída de sistemas heterogêneos (MELLO, 2005).

Isso acontece pois há uma preferência pelo uso de políticas de recuperação sob demanda como o *garbage collection*, o *fossil collection* e protocolos baseados em rollbacks (*rollback-based protocols*) ao invés de mecanismos que limitam o armazenamento de *checkpoints*. Políticas de recuperação sob demanda são políticas que são executadas de acordo com uma necessidade. Por exemplo, quando a memória estiver cheia, uma política de recuperação de memória sob demanda pode ser invocada.

Uma abordagem comum de *garbage collection* é identificar o último conjunto consistente de *checkpoints* e remover todas as informações de eventos que ocorreram antes disso (WANG, 1993). Entretanto em casos especiais (i.e., quando o último estado consistente é o estado inicial), essa abordagem não conseguirá remover nenhum *checkpoint*, levando a um transbordamento de memória (LIU; CHEN, 1999).

O *fossil collection* tem uma abordagem parecida com o *garbage collection*. A implementação mais simples compara o *timestamp* da informação com o valor do GVT. Assim toda informação ocorrida antes do GVT pode ser removida (YOUNG; WILSEY, 1996). Esse método tem o mesmo problema que o *garbage collection*, correndo o risco de não conseguir excluir nenhum *checkpoint*.

Quando as abordagens acima não obtém sucesso na recuperação de memória, os protocolos baseados em *rollbacks* podem ser invocados. Estes protocolos induzem os processos a executarem *rollbacks*, gerando informações inúteis, tais como *logs* de mensagens e *checkpoints* obsoletos, que podem ser apagadas liberando a memória. Porém, se essa abordagem também não liberar memória, a simulação pode entrar em colapso (DAS; FUJIMOTO, 1994).

Ao contrário de abordagens que utilizam políticas de recuperação sob demanda, que

ainda correm o risco de ficar sem memória, abordagens que limitam o espaço de memória (i.e., uso de memória virtual) passam uma impressão de memória infinita, devido as políticas de substituição (similares a protocolos de memória cache em sistemas operacionais) (MITTRA, 1995).

A dificuldade está justamente em criar/adaptar esses mecanismos que limitem o espaço de memória utilizado pelos *checkpoints* de cada processo e encontrar técnicas de substituição desses *checkpoints*, para quando o espaço de memória definido atingir um estado crítico. Geralmente é complexo implementar soluções desse tipo que não tenham impacto negativo no desempenho da simulação.

Para contribuir com soluções para o problema citado acima, este trabalho apresenta uma abordagem que limita o espaço de memória utilizado por *checkpoints*. Através de simulações de *rollbacks* reais, esta abordagem identifica os *checkpoints* que podem ser úteis para futuras recuperações de falhas e exclui os demais *checkpoints*.

A implementação dessa abordagem se torna útil pois modelos grandes ou complexos geram uma quantidade grande de informações, que podem se tornar inúteis com o passar do tempo. Além disso, métodos que não limitam o espaço de armazenamento não impedem um eventual *overflow* da memória.

Com a implementação da política criada foi possível obter uma diferença no tamanho do *buffer* de *checkpoints* com até 34 vezes menos checkpoints em relação a versão anterior. A manutenção do espaço de armazenamento limitado permite que grandes simulações executem até o fim, sem risco de parada por falta de memória.

O restante deste trabalho está organizado da seguinte forma: No capítulo 2 é apresentada uma contextualização do problema; o capítulo 3 apresenta alguns trabalhos relacionados; no capítulo 4 é apresentado o desenvolvimento do trabalho, onde são apresentados algoritmos que resolvem o problema encontrado e também são apresentados os resultados alcançados com os casos de teste; e por fim, a conclusão é apresentada no capítulo 5.

2 SIMULAÇÃO DISTRIBUÍDA

2.1 Simulação

Simulação é uma maneira de imitar um sistema real através de modelos. Ou seja, através de recursos computacionais é possível representar um sistema e executar experimentos a fim de entender como um sistema funciona ou deveria funcionar, ajudando na tomada de decisões (PIDD, 1994).

Dentre as vantagens dessa abordagem estão: A possibilidade de verificar o sistema antes de implementar, entender como um sistema funciona, testar novas regras ou hipóteses, identificar erros ou gargalos, analisar sistemas de longa duração em um tempo curto e fazer testes em sistemas caros ou perigosos (SHANNON, 1998) (LAW; KELTON, 1997).

Quando uma simulação se torna grande ou complexa demais, técnicas de simulação distribuída podem sem utilizadas, subdividindo uma simulação em vários processos paralelos ou distribuídos.

2.2 Simulação Distribuída

Uma simulação pode ter seus componentes subdivididos em vários processos que serão executados paralelamente, assim, sendo nomeada de simulação distribuída. Como vários componentes vão executar ao mesmo tempo isso acaba diminuindo o tempo da simulação. Além disso, a distribuição da simulação também permite a heterogeneidade dos componentes (*p.ex.* máquinas de diferentes arquiteturas) (FUJIMOTO, 2001).

Como cada componente é um processo, eles interagem entre si por troca de mensagens. Visto que a simulação não é mais executada em uma única máquina, cada componente possui uma noção de tempo local (*LVT - Local Virtual Time*) para que seus eventos sejam sincronizados com os eventos de outros componentes, garantindo uma ordem correta de execução (PREISS; MACINTYRE; LOUCKS, 1992). Esses tempos locais dos componentes também auxiliam no cálculo do tempo virtual global (*GVT - Global Virtual Time*).

Se tratando de sincronização de eventos há duas abordagens: a síncrona (conservadora) e a assíncrona (otimista).

2.2.1 Simulação Síncrona (Conservadora)

Em uma abordagem conservadora o algoritmo de sincronização deve tomar cuidado para que violações de tempo (*LCC - Local Constraint Causality*) não ocorram. Uma violações de tempo (*LCC*) ocorre quando um componente recebe um evento com tempo menor que o seu tempo atual (*LVT*). Portanto, o algoritmo deve determinar quando é seguro um componente processar um evento, garantindo que nenhum evento com tempo menor será recebido depois que o componente avançou no tempo (FUJIMOTO, 2001).

Nessa estratégia os processos ficam ociosos, devido a sincronia com os outros processos, levando a uma perda de eficiência (FERSCHA; TRIPATHI, 1994).

2.2.2 Simulação Assíncrona (Otimista)

A abordagem assíncrona também é conhecida como otimista, pois avança no tempo sem se preocupar com os eventos solicitados pelos demais processos.

Essa estratégia permite que violações de tempo (*LCC*) ocorram, tornando a simulação inconsistente. Portando, para esses métodos se tornam necessários algoritmos de recuperação de falhas.

Na figura 2.1 é ilustrada uma violação de tempo (*LCC*) onde o processo PL0 solicita a execução de um evento por PL1 com um tempo menor (40) que o atual de PL1 (60), criando uma inconsistência na simulação.

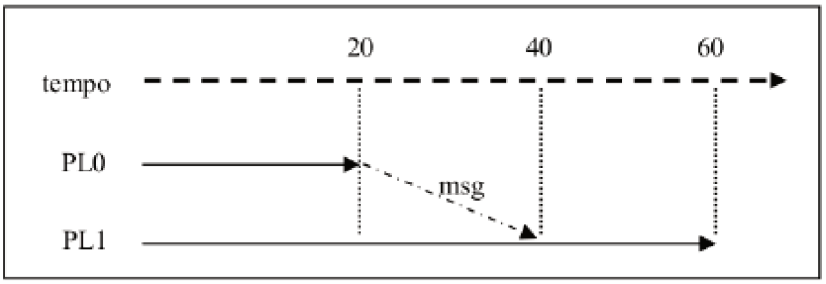


Figura 2.1: LCC - Violação de tempo

2.3 Gerenciamento de Falhas

Nas simulações assíncronas são necessários algoritmos de recuperação de falhas. Esses algoritmos, chamados de protocolos de *rollback*, são responsáveis por retroceder uma simulação

à um estado global consistente (livre de falhas), não sendo necessário iniciar a simulação do começo.

O estado global é um conjunto dos estados de cada processo participante do sistema (ELNOZAHY et al., 2002). Um estado global é dito consistente se não existem mensagens órfãs. Uma mensagem é órfã quando o evento que recebeu está salvo, mas o evento que enviou se perdeu (KUMAR; CHAUHAN; KUMAR, 2010).

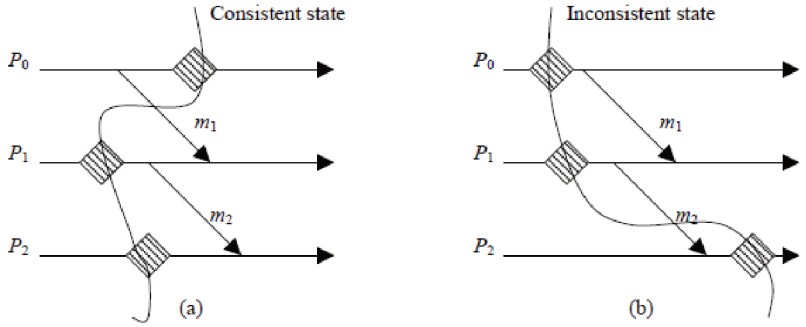


Figura 2.2: Estado global consistente e inconsistente (ELNOZAHY et al., 2002)

A figura 2.2 ilustra os estados globais. Em 2.2 (a) o estado é consistente e em 2.2 (b) o estado é inconsistente, pois a mensagem *m2* é órfã.

2.4 *Checkpoints*

Checkpointing é uma técnica bem sucedida usada para recuperação de falhas em sistemas distribuídos (KUMAR; CHAUHAN; KUMAR, 2010).

Um *checkpoint* é uma cópia do estado da aplicação que pode ser usado para reiniciar a execução da simulação em caso de falhas (SATO et al., 2012). Caso uma violação de tempo ocorra, um *rollback* restaura a simulação para o estado consistente mais recente.

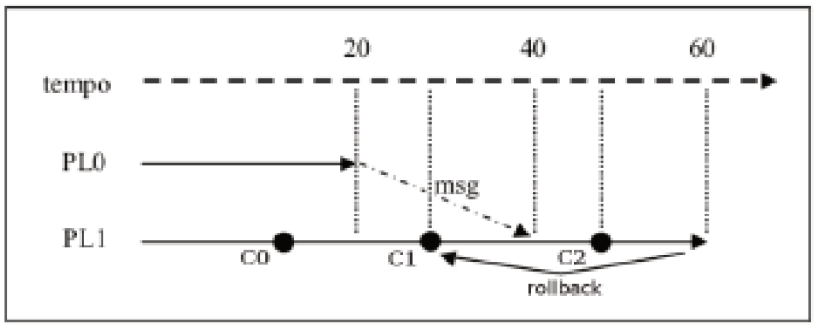


Figura 2.3: Exemplo de *checkpointing*

A figura 2.3 mostra um exemplo da utilização de *checkpoints*. O processo PL0 envia

uma mensagem com tempo menor (40) que o atual de PL1 (60) causando uma violação de tempo e forçando um *rollback* para o *checkpoint* C1 que possui tempo anterior da mensagem enviada por PL0, garantindo a ordem correta da execução.

Técnicas de *rollback* baseadas em *checkpoints* podem ser classificadas como *checkpoints* coordenados, não-coordenados e induzidos por comunicação.

Checkpoints coordenados sincronizam a criação de *checkpoints*, ou seja, quando um processo cria um *checkpoint* ele solicita para que todos os processos relevantes criem *checkpoints* também (CAO; SINGHAL, 1998). A desvantagem dessa abordagem é a grande quantidade de mensagens de controle trocadas.

A não-coordenação de *checkpoints* permite que cada processo decida quando tomar *checkpoints*. A vantagem dessa técnica é a diminuição do *overhead* causado pela troca de mensagens, visto que cada processo toma *checkpoints* quando for mais conveniente, sem depender dos outros processos. Entre as desvantagens dessa abordagem estão a possibilidade de efeito dominó em caso de *rollback* e a criação de *checkpoints* inúteis (ELNOZAHY et al., 2002).

Checkpoints induzidos por comunicação combinam as duas técnicas citadas acima. Nesse protocolo os processos criam dois tipos de *checkpoints*, locais e forçados. *Checkpoints* locais podem ser criados independentemente pelos processos, enquanto os *checkpoints* forçados são criados baseados nas informações trocadas por mensagens entre os processos. Essa abordagem diminui o *overhead* criado por mensagens de controle e evita o efeito dominó (ELNOZAHY et al., 2002).

2.5 *Checkpoints* Inúteis

Quando os componentes criam *checkpoints* de forma não-coordenada, pode ser que alguns desses *checkpoints* nunca serão usados, ou seja, não farão parte de um estado global consistente, sendo considerados inúteis.

Em (PARIZOTTO; MELLO, 2017) são observados três tipos de *checkpoints* inúteis:

- *Checkpoints* inalcançáveis: Nunca são restaurados por operações de *rollback* (Figura 2.4 C0).
- *Checkpoints* insuficientes: *Checkpoints* que são restaurados, mas não fazem parte de um estado global consistente (Figura 2.4 C1).

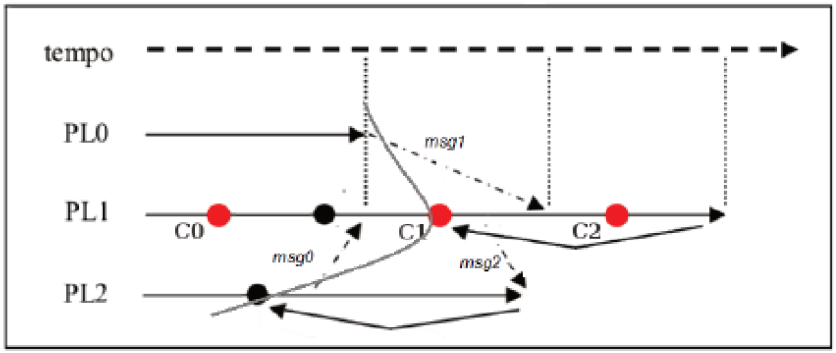


Figura 2.4: Exemplo de *checkpoints* inúteis

- *Checkpoints* inconsistentes: *Checkpoints* que armazenam um estado com *LVT* maior que o *LVT* atual do processo (Figura 2.4 C2).

2.6 Eliminação de *Checkpoints*

Grandes simulações exigirão uma grande quantidade de *checkpoints* para controle de falhas. Com o andamento da simulação alguns *checkpoints* podem se tornar obsoletos, ou seja, não serão mais úteis para uma execução de *rollback*. *Checkpoints* obsoletos (inúteis) se tornam um problema, pois além de não contribuírem para a recuperação da simulação em caso de falhas, eles consomem espaço de armazenamento.

A partir deste problema surgem os algoritmos de recuperação de memória sob demanda, que quando executados recuperam a memória ocupada por informações inúteis, tais como *checkpoints* e *logs* de mensagens. Entre os mais conhecidos algoritmos de recuperação estão o *Garbage Collection* e o *Fossil Collection*. O capítulo dos trabalhos relacionados comenta o funcionamento deles.

2.7 DCB

O DCB é uma arquitetura de simulação onde é possível criar e executar simulações de modelos distribuídos e heterogêneos.

O DCB é composto por quatro módulos principais: o *gateway*, o Núcleo do DCB (*DCB Kernel - DCBK*), o Expedidor do DCB (*DCB Sender - DCBS*) e o Receptor do DCB (*DCB Receiver - DCBR*).

Devido a heterogeneidade dos componentes, é necessário uma interface de comunicação entre o elemento e os demais módulos. No DCB o responsável por fazer essa interface é o

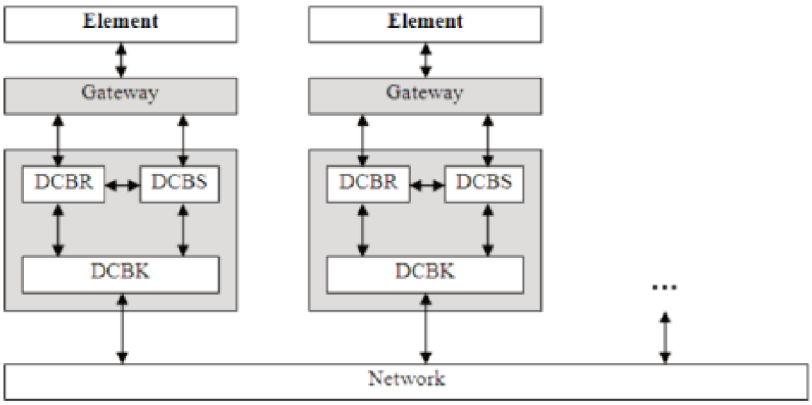


Figura 2.5: Arquitetura do DCB

gateway. Ou seja, o *gateway* é capaz de traduzir as informações de um formato de origem para um formato de destino de acordo com a necessidade.

O núcleo do DCB (*DCBK*) é o serviço que controla a troca de mensagens. Além disso, ele mantém um único valor do tempo virtual global (*GVT*) para todos os nodos.

O *DCBR* é responsável pelas mensagens recebidas de outros componentes. Ele decodifica as mensagens recebidas e participa do gerenciamento de tempo e simulação.

O *DCBS* é o serviço de gerenciamento de mensagens enviadas pelo componente. Em conjunto com o *DCBR*, também participa do gerenciamento de tempo.

A sincronização das mensagens no DCB é feita através do tempo de evento (*timestamp*). O *timestamp* é enviado junto da mensagem e indica a ordem que o evento deve ser executado.

O DCB permite que os componentes avancem no tempo de modo assíncrono ou síncrono, suportando assim um modelo de sincronização híbrido.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta um estudo de métodos propostos para reduzir a média de *checkpoints* criados em simulações distribuídas e técnicas de recuperação sob demanda de espaço ocupado por informações desnecessárias, tais como *logs* de mensagens e *checkpoints* obsoletos. Também serão discutidas algumas estratégias existentes de gerenciamento de memória limitada e alguns benefícios na utilização destes métodos.

Em (WANG, 1993) uma abordagem comum do *garbage collection* é citada usando a ideia de *checkpoints* obsoletos, onde o mais recente conjunto de *checkpoints* consistente é identificado. Este conjunto de *checkpoints* consistente, também chamado de *recovery line* é encontrado através de um grafo de dependência montado através das mensagens enviadas e recebidas pelos processos. Os vértices do grafo representam os *checkpoints* e as arestas representam uma dependência entre esses *checkpoints*. Depois disso, a propagação de um *rollback* é executada no grafo, determinando uma linha de recuperação segura (*recovery line*), onde todas as informações referentes a eventos que ocorreram antes disso são descartadas. A vantagem dessa abordagem é a facilidade de implementação.

Liu e Chen em (LIU; CHEN, 1999) encontram um problema no algoritmo citado por Wang *et al.*, provando facilmente que em casos especiais a abordagem comum do *garbage collection* não conseguirá apagar nenhum checkpoint. Um caso especial citado é o do último estado consistente de checkpoints ser o estado inicial. Liu e Chen também propõem métricas para encontrar *checkpoints* úteis e descartar com segurança os *checkpoints* obsoletos ou inválidos. A abordagem deles também usa um grafo de dependência, porém identificando todos os *checkpoints* que podem ser úteis para um *rollback*. Desta forma não somente os *checkpoints* criados antes do conjunto consistente serão apagados, mas em caso de rollback, os que estão depois do conjunto consistente poderão ser eliminados, visto que não são mais úteis.

(MITTRA, 1995) propõe o uso do conceito de memória virtual no simulador *Verilog-XL*. O esquema proposto funciona de maneira parecida com os protocolos de memória cache. Se o endereço de memória necessário já está carregado na memória virtual, ele pode ser acessado diretamente, entretanto se ele não está carregado, é necessário fazer a carga antes de usá-lo. Esta proposta fornece a capacidade de ter, teoricamente, um espaço de memória infinito. No entanto, essa abordagem usa *swap* entre a memória principal e o armazenamento secundário, podendo deixar a simulação mais lenta.

Em (YOUNG; WILSEY, 1996) os autores citam uma implementação tradicional do *fossil collection*, onde o *timestamp* do item (*checkpoint*, log de mensagem, etc.) é comparado com o GVT. Assim somente as informações necessárias para fazer um rollback para o GVT são mantidas, e todas as informações anteriores ao GVT podem ser removidas. Os autores também apresentam uma nova técnica chamada *Optimistic Fossil Collection* (OFC) que estabelece modelos probabilísticos e usa um fator de risco definido pelo usuário para decidir se um item pode ser recuperado. Basicamente, se a probabilidade for menor que o fator de risco o item pode ser eliminado. Assim como os próprios autores reconhecem, uma vez que a identificação é probabilística, há uma chance de ocorrerem erros.

Em (CAROTHERS; BAUER; PEARCE, 2000) é apresentado um novo mecanismo para simulações distribuídas chamado ROSS (*Rensselaer's Optimistic Simulation System*). Nesse mecanismo cada processador aloca um conjunto separado de memória para cada processador remoto, assim a memória é compartilhada apenas por um par de processadores. Quando a aplicação requisita memória, o processador de origem determina qual conjunto de memória do processador de destino será usado. Se o conjunto de memória livre no processador de destino estiver vazio o evento não é escalonado. Quando um evento termina sua execução, o cálculo do GVT é iniciado e protocolos de recuperação de memória retornam a memória livre para o conjunto de memória do processador proprietário. Uma vantagem dessa abordagem é que o excesso de otimismo é evitado, uma vez que a execução de processos em um processador é limitada pela sua memória livre.

Em (PARIZOTTO; MELLO, 2017) são propostas métricas para reduzir a média de *checkpoints* criados por meio da identificação de *checkpoints* inúteis. A estratégia é baseada em técnicas não coordenadas, não exigindo troca de mensagens exclusivamente para controle, mas sim usando o histórico de mensagens para decidir probabilisticamente se um *checkpoint* pode se tornar inútil. Esta solução foi implementada e analisada no DCB (*Distributed Simulation Backbone*) (MELLO, 2005).

4 DESENVOLVIMENTO

Este capítulo mostra uma solução para o problema da falta de memória causada pelo armazenamento de *checkpoints*. Serão apresentadas a descrição da solução e do algoritmo, a implementação do algoritmo no DCB, o modelo utilizado na simulação e por fim serão apresentados os estudos de caso e os resultados encontrados com a implementação em comparação com a versão anterior implementada por (PARIZOTTO; MELLO, 2017);

4.1 Contextualização da Solução

Grandes simulações produzem muitas informações que podem se tornar desnecessárias com o passar do tempo. Essas simulações podem parar por falta de memória, já que trabalhos relacionados excluem essas informações mas não limitam o espaço de armazenamento. Desta forma, este trabalho apresenta o desenvolvimento de uma estratégia que limita a memória usada por *checkpoints* para evitar que a simulação entre em colapso.

A ideia surgiu da análise da estratégia citada em (WANG, 1993) e (LIU; CHEN, 1999), onde um grafo de dependência entre os *checkpoints* é criado para, posteriormente, simular um *rollback* sobre esse grafo, identificando os *checkpoints* inúteis.

Na solução proposta não foi criado um grafo, mas foram analisadas as mensagens enviadas pelos processos, para simular um *rollback* e o efeito dominó, com o propósito de descobrir o ponto máximo que um *rollback* pode alcançar. A simulação de um *rollback* é a execução dos mesmos passos que um *rollback* real executaria, mas sem modificar os componentes. O ponto máximo de um *rollback* é definido pelos *checkpoints* que indicam até onde um *rollback* pode chegar, executando o efeito dominó. Esse ponto máximo composto por um *checkpoint* de cada componente é chamado de *recovery line*.

É importante observar que a chamada da simulação do *rollback* pode retornar resultados diferentes para cada componente. Portanto a simulação do *rollback* deve ser chamada para cada componente, e deverão ser mantidos *N recovery lines* onde *N* é o número de componentes. Desta forma, em uma simulação poderão existir até *N * N checkpoints* necessários para recuperações.

Após a execução das simulações de *rollbacks* e a definição das *recovery lines*, todos os *checkpoints* úteis para recuperações de falhas são identificados, o que significa que os demais *checkpoints* podem ser eliminados com segurança, já que se tornam inalcançáveis ou insufici-

entes.

A figura 4.1 mostra as possíveis *recovery lines* de 2 componentes *A* e *B*. Se uma simulação de falha for iniciada por *A*, como ele não mandou nenhuma mensagem para *B* depois do último checkpoint, o *rollback* alcançará no máximo os últimos *checkpoints*, assim a *recovery line* de *A* são os *checkpoints* 2 e 4. Caso a simulação de falha for iniciada por B, a última mensagem causa um efeito dominó, fazendo com que A também execute um *rollback* até que não existam mais mensagens órfãs. Desta forma a *recovery line* de *B* são os *checkpoints* 1 e 3.

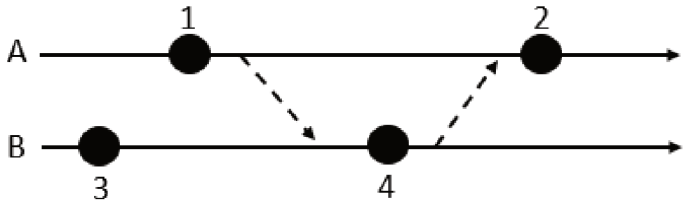


Figura 4.1: Possíveis *Recovery Lines*

Como diferencial das demais abordagens, a técnica desenvolvida contém uma limitação de *checkpoints* criados. Quando o número de *checkpoints* armazenados chegar ao limite determinado no modelo, o algoritmo de busca das *recovery lines* é invocado, podendo, após isso, excluir os *checkpoints* inalcançáveis e insuficientes, evitando que a simulação entre em colapso por falta de memória. O limite de *checkpoints* armazenados deve ser determinado considerando o número mínimo de *N checkpoints*, onde *N* é o número de componentes. Além desse número mínimo, deve ser considerada uma margem para que o algoritmo possa trabalhar.

4.2 Algoritmo

Por se tratar de um modelo distribuído, cada componente controla somente os seus *checkpoints*. Desta forma, quando for verificado que o *buffer* de *checkpoints* do componente está em estado crítico, ou seja, quase cheio, o algoritmo de busca da *recovery line* será chamado. A verificação do estado do *buffer* é feita a cada *checkpoint* armazenado.

Assim como já citado acima, cada componente pode gerar uma *recovery line* diferente. Desta forma, quando um componente verificar que seu *buffer* de *checkpoints* está chegando a um estado crítico, ele deve sinalizar a todos os demais componentes para buscarem suas *recovery lines* também.

Quando for chamada a busca da *recovery line*, o processo simulará uma falha no seu

tempo atual de execução (LVT). Desta forma o maior *checkpoint* com *timestamp* menor que o tempo atual será buscado. Após isso são verificadas as mensagens enviadas pelo componente para obter aquelas com *timestamp* maior que o do *checkpoint*, ou seja, mensagens que se tornarão órfãs. Cada mensagem órfã deve ser reenviada para o componente de destino (antimensagem), como forma de avisar que o componente de destino também deverá realizar um *rollback*, isso é chamado de efeito dominó. Quando o componente de destino receber uma antimensagem, ele realizará um *rollback* para o maior *checkpoint* com *timestamp* menor que o *timestamp* da antimensagem recebida. Este último *rollback* pode causar novas mensagens órfãs e o algoritmo só terminará quando não houverem mais antimensagens a serem enviadas.

Como os componentes executarão de forma distribuída, cabe a cada componente armazenar os seus *checkpoints* que farão parte de uma *recovery line*. Considerando novamente que podem existir *N recovery lines*, onde *N* é o número de componentes, um componente pode armazenar até *N checkpoints*.

Com o fim do cálculo das *recovery lines* todos os *checkpoints* úteis para *rollbacks* foram descobertos e os demais podem ser excluídos.

Abaixo estão alguns pseudo-códigos com o algoritmo descrito acima.

A função *PropagarBuscaRecoveryLine* propaga a chamada da busca da *recovery line* para todos os componentes do modelo. Essa função é chamada sempre que um componente precisa buscar sua *recovery line*, sinalizando aos demais componentes para fazerem o mesmo.

Algoritmo 1: PropagarBuscaRecoveryLine

```
   Entrada: componentes
1  início
2  |  repita
3  |  |  BuscarRecoveryLine(componente.LVT,
   |  |   componente.checkpointsArray,
   |  |   componente.bufferSentMessages, componente)
4  |  até até ler todos os componentes;
5  fim
```

O método *VerificaSituacaoListaCheckpoints* recebe como entrada dois inteiros. São eles os números de *checkpoints* onde devem ser chamadas a busca da *recovery line* e a exclusão dos *checkpoints*. Ou seja, analisando a lista de *checkpoints* criados, quando o tamanho dessa lista atingir os números passados na entrada da função, o cálculo da *recovery line* e exclusão dos *checkpoints* são invocados.

Algoritmo 2: VerificaSituacaoListaCheckpoints

```
Entrada: numeroCheckpointsChamaRecoveryLine,
         numeroCheckpointsExcluiRecoveryLine
1 início
2   if checkpointsArray.size() = numeroCheckpointsChamaRecoveryLine then
3     BuscarRecoveryLine(this.LVT, checkpointsArray,
       bufferSentMessages, this)
4   if checkpointsArray.size() = numeroCheckpointsExcluiRecoveryLine then
5     ExcluirCheckpointsForaRecoveryLine(checkpointsArray,
       checkpointsRecoveryLine)
6 fim
```

O método *BuscarRecoveryLine* é o responsável pela execução do algoritmo proposto acima. A função recebe como entrada quatro parâmetros: *timestampFalha*, *checkpointsArray*, *bufferSentMessages* e *componente*. O *timestampFalha* indica o tempo em que a falha deve ser simulada. Ele recebe o tempo atual se a simulação da falha está iniciando, ou o *timestamp* de uma antimensagem, caso algum outro componente está solicitando um efeito dominó. O parâmetro *checkpointsArray* diz respeito a lista de *checkpoints* armazenados até então. Já o *bufferSentMessages* são as mensagens enviadas pelo componente que está simulando a falha. Por fim, o *componente* recebe qual componente está solicitando o armazenamento de um *checkpoint* para uma possível *recovery line*, ou seja, qual elemento iniciou a simulação de falha. Desta forma, quando as antimensagens são enviadas, elas devem carregar consigo o elemento que iniciou a simulação do retrocesso.

Durante sua execução o algoritmo *BuscarRecoveryLine* busca o maior *checkpoint* menor que o *timestampFalha* (*getCheckpoint*), procura por todas as mensagens com *timestamp* maior que o *checkpoint* encontrado (*getMensagensOrfas*) e envia uma antimensagem para cada mensagem encontrada (*EnviarAntiMessageRecoveryLine*). As antimensagens enviadas servem para avisar para o componente de destino que agora ele deve simular uma falha a partir do *timestamp* dessa antimensagem, e também enviar o componente que está iniciando a simulação da falha. Após todas as antimensagens serem enviadas, o *checkpoint* encontrado é armazenado com a identificação do componente que solicitou, pois ele pode fazer parte de uma *recovery line* (*AddOuSubstituiCheckpointRecoveryLine*). Caso um efeito dominó seja simulado, e um *checkpoint* já está armazenado com a identificação do *componente* passado por parâmetro, o checkpoint é substituído.

Algoritmo 3: BuscarRecoveryLine

```
   Entrada: timestampFalha, checkpointsArray,
            bufferSentMessages, componente
1  início
2  |  checkpoint = getCheckpoint(timestampFalha,
   |   checkpointsArray)
3  |  mensagensOrfas = getMensagensOrfas(checkpoint,
   |   bufferSentMessages)
4  |  repita
5  |  |  EnviarAntiMessageRecoveryLine(componente)
6  |  até mensagensOrfas != Empty;
7  |  AddOuSubstituiCheckpointRecoveryLine(checkpoint,
   |   componente)
8  fim
```

O algoritmo *RecebeAntiMensagem* é o responsável por chamar a *BuscaRecoveryLine* quando recebe uma antimensagem. O único parâmetro recebido é a antimensagem que é enviada por um outro componente, quando este executa uma simulação de efeito dominó.

Este método somente executa a busca da *recovery line* passando o *timestamp* da mensagem recebida, os *checkpoints* armazenados no componente que está executando, as mensagens enviadas até então pelo componente que está executando e o componente que iniciou a simulação que é propagado junto com a mensagem.

Algoritmo 4: RecebeAntiMensagem

```
   Entrada: mensagem
1  início
2  |  BuscaRecoveryLine(mensagem.Timestamp,
   |   checkpointsArray, bufferSentMessages,
   |   mensagem.Componente)
3  fim
```

O último algoritmo é o *ExcluiCheckpointsForaRecoveryLine*. Este tem a função de remover todos os checkpoints inúteis, ou seja, aqueles que não pertencem a nenhuma *recovery line*. Os parâmetros passados são a lista de *checkpoints* armazenados e a lista de *checkpoints* que pertencem a uma *recovery line*. O algoritmo simplesmente passa todos os *checkpoints* armazenados, excluindo aqueles que não estão na lista de *checkpoints* que pertencem a uma *recovery line*.

Algoritmo 5: ExcluirCheckpointsForaRecoveryLine

```
   Entrada: checkpointsArray, checkpointsRecoveryLine
1  início
2  |  repita
3  |  |  Pega checkpoint
4  |  |  if checkpointsRecoveryLine não contém checkpoint then
5  |  |  |  checkpointsArray.Remove(checkpoint)
6  |  até ler todos em checkpointsArray;
7  fim
```

4.3 Sincronização e Checkpointing no DCB

O DCB suporta três tipos de sincronização: síncrona, assíncrona e *notime*, onde cada componente pode assumir um tipo de sincronização e ter seu tempo incrementado de acordo com o modelo.

Componentes síncronos avançam a execução quando não houver mais mensagens para o seu tempo. Os síncronos avançam de forma otimista, sem se preocupar com violações de tempo, e são amparados por mecanismos de *rollback*. Já os componentes *notime* ordenam as mensagens de acordo com a sua chegada, e executam estas sem se preocupar com o tempo.

O DCB cria *checkpoints* de forma não-coordenada, ou seja, não troca mensagens de controle para a criação de *checkpoints*, diminuindo o *overhead*.

O algoritmo é baseado nas mensagens recebidas, onde são criadas métricas probabilísticas que decidem se um *checkpoint* será útil. Caso não for útil o *checkpoint* não é criado.

O protocolo de *checkpointing* não possui um controle de memória limitada para *checkpoints*, possibilitando uma interrupção da simulação por falta de memória.

4.4 Implementação do algoritmo no DCB

Com a finalidade de provar a funcionalidade do algoritmo, este foi implementado no DCB. A maioria das alterações foi feita na classe *Chat*, pois é onde estão implementados os algoritmos de *checkpointing* do DCB.

A função *VerificaSituacaoListaCheckpoints* foi implementada na classe *Chat* do DCB, pois é onde o *buffer* de checkpoints está implementado. Caso o *buffer* de *checkpoins* esteja em um estado crítico, as funções de propagação da busca da *recovery line* e exclusão de *checkpoints* são chamadas.

O algoritmo *PropagarBuscaRecoveryLine* foi implementado na classe *EDCB* com o

nome *MensagemInicioRecoveryLine*, mandando uma mensagem para cada componente, indicando que este deve iniciar a busca da *recovery line*. A classe *EDCB* correspondente o módulo *DCBS* do DCB. A função foi implementada nessa classe, pois é onde são controladas todas as mensagens enviadas pelo componente.

O método *BuscarRecoveryLine* também foi implementado na classe *Chat*. Nesta função são enviadas as antimensagens e armazenados os *checkpoints* das *recovery lines*. A função foi implementado na classe *Chat* pois é onde os *checkpoints* pertencentes a alguma *recovery line* são armazenados.

A função que fica esperando uma antimensagem foi implementada na classe *GatewayMPI* com o nome *ProtocolConverter*. Ao receber uma antimensagem o método *BuscarRecoveryLine* é chamado. A classe *GatewayMPI* é responsável pela tradução das mensagens enviadas e recebidas, por esse motivo a leitura de uma antimensagem é implementada nela.

Por fim o algoritmo *ExcluirCheckpointsForaRecoveryLine* foi implementado na classe *Chat*, que é onde estão armazenados o *buffer* de *checkpoints* e os *checkpoints* das *recovery lines*.

4.5 Modelo

O modelo utilizado para executar os estudos de caso consiste em 5 componentes evoluindo no tempo de forma assíncrona. Foi utilizado o mesmo modelo utilizado em (PARIZOTTO; MELLO, 2017), para que pudesse ser feito uma comparação entre os resultados.

A figura 4.2 mostra o modelo de troca de mensagens. Os rótulos nas arestas representam a probabilidade de uma mensagem ser enviada em um intervalo de 1 segundo. Desta forma a troca de mensagens entre os componentes se torna probabilística.

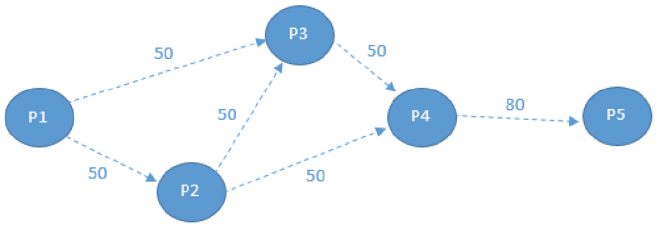


Figura 4.2: Modelo de troca de mensagens

Para efeito de testes os componentes aumentam seus tempos de forma diferente. O processo 1 aumenta seu LVT em 75 unidade de tempo por evento executado, o 2 aumenta 100, o 3 aumenta 150, o 4 aumenta 200 e o 5 aumenta 250. Desta forma *rollbacks* são forçados,

causados por mensagens enviadas do componente 2 para os componentes 3 e 4, e mensagens enviadas do componente 3 para o componente 4.

As simulações foram executadas em único computador: Intel®Core™ i5-5200 CPU @ 2.20GHz com sistema operacional Ubuntu 18.04. Foram executadas 10 vezes de 100.000 unidades de tempo, com a versão de (PARIZOTTO; MELLO, 2017) e com a versão aqui proposta.

O espaço disponível para o armazenamento de *checkpoints* foi limitado em 10 unidades, onde com 70% é chamado o cálculo da *recovery line* e com 90% é chamado a exclusão dos *checkpoints* desnecessários.

Neste modelo foram definidas as seguintes variáveis para posterior análise dos resultados:

- Tempo total de execução
- Total de mensagens enviadas
- Total de mensagens recebidas
- Total de *checkpoints* criados
- Total de *checkpoints* mantidos

O tempo total de execução foi medido para que pudessem ser comparados os tempos de execução de cada componente na implementação antiga e na nova. O total de mensagens enviadas e recebidas foi medido para poder comparar a diferença no número de mensagens trocadas entre cada componente nas duas implementações. Por fim, foram contados os *checkpoints* criados e mantidos, para provar que o número de *checkpoints* armazenados diminuiu na nova versão. Todas estas variáveis foram analisadas para comparar os ganhos em espaço de armazenamento com as perdas em processamento e tempo de simulação.

4.6 Estudo de caso e resultados

As figuras abaixo apresentam dados de três componentes executados na versão implementada por (PARIZOTTO; MELLO, 2017) e na versão apresentada aqui utilizando o modelo acima citado.

Na figura 4.3 é apresentado o tempo de execução de cada componente. É possível observar que os componentes 6 e 7 sofreram um aumento no tempo total de simulação, devido ao

processamento dos algoritmos de busca de *recovery lines* e exclusão de *checkpoints*. O aumento na troca de mensagens de controle (figuras 4.4a e 4.4b) também afeta o tempo de execução. O componente 5 não sofreu mudança no tempo pois no modelo usado ele somente envia mensagens para os outros componentes. Desta forma o componente 5 não executa *rollbacks*.

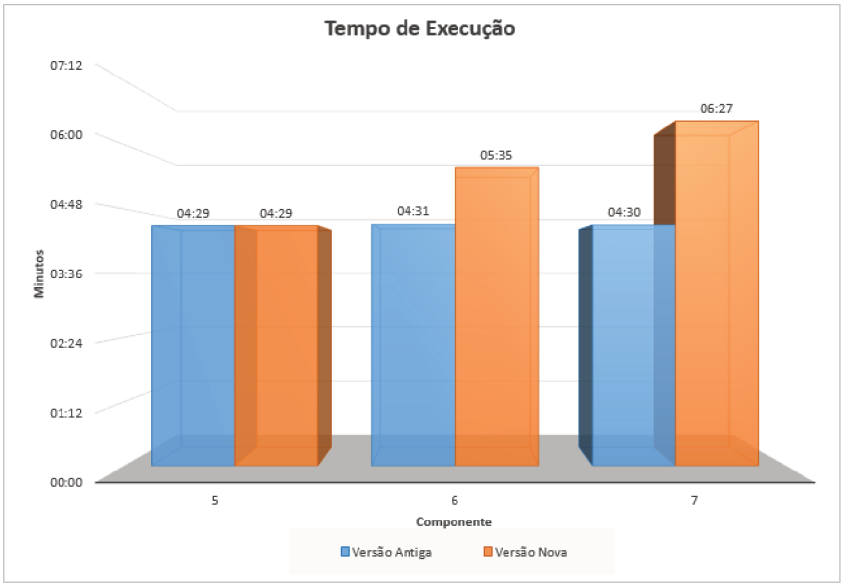


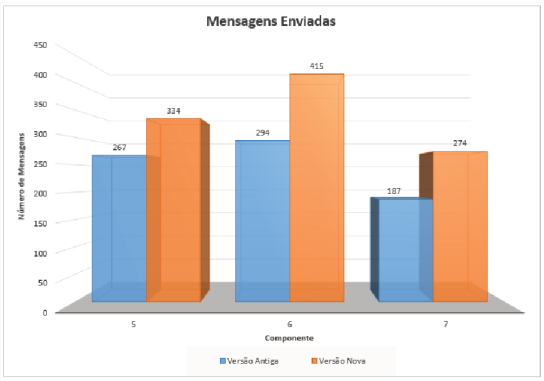
Figura 4.3: Tempo total de execução da simulação

A quantidade de mensagens trocadas também aumenta, devido a necessidade de um *broadcast* quando um componente inicia uma simulação de falha para avisar à todos os componetes que eles também devem iniciar a busca das suas *recovery lines*. Além disso, ainda existem as antimensagens trocadas durante a simulação da falha, para executar o efeito dominó. As figuras 4.4a e 4.4b mostram a diferença do número de mensagens trocadas entre os componente das duas versões. Nas figuras são consideradas todas as mensagens enviadas e recebidas, não somente as de controle.

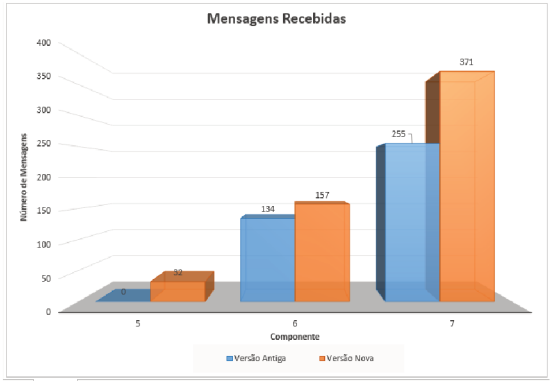
Devido ao aumento do tempo de execução, o número de *checkpoitns* criados também aumenta, uma vez que os *checkpoints* são criados de forma probabilística.

Conforme a figura 4.5b o número de *checkpoints* mantidos diminuiu drasticamente com a implementação das *recovery lines*, nunca passando de 10 *checkpoints*, que foi o limite definido no modelo.

Como no modelo escolhido existem 5 componentes, cada componente pode manter até 5 *checkpoints* úteis, um para cada *recovery line*. No modelo foram definidas 10 unidades para que o algoritmo tivesse uma margem para trabalhar. Este não é um limite significativo para simulações grandes, entretanto foi o suficiente para verificar e constatar o correto funcionamento

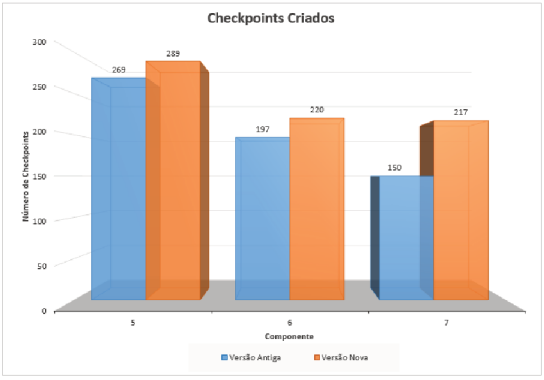


(a) Mensagens Enviadas

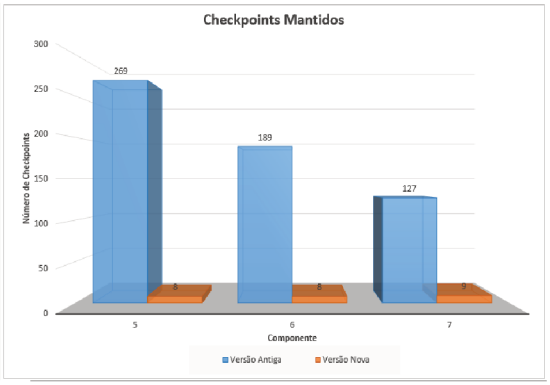


(b) Mensagens Recebidas

Figura 4.4: Total de mensagens trocadas



(a) *Checkpoints* criados



(b) *Checkpoints* mantidos

Figura 4.5: Total de *checkpoints*

do algoritmo no modelo proposto.

A quantidade de *rollbacks* reais e o comprimento desses *rollbacks* não mudaram. O comprimento de um *rollback* é a diferença entre o LVT que o componente tinha e o LVT que o componente passou a ter depois do *rollback*. Essa mudança não ocorre pois as *recovery lines* são usadas somente para encontrar os *checkpoints* úteis e não para executar *rollbacks* reais. Desta forma, se *checkpoints* forem criados depois da chamada da exclusão dos *checkpoints* inúteis e esses *checkpoints* forem úteis para uma recuperação de falha, eles serão usados mesmo não sendo parte de uma *recovery line*. Por outro lado, como uma *recovery line* é descoberta simulando um *rollback*, um *rollback* real nunca voltará para um tempo inferior a linha de recuperação.

Diante do cenário utilizado para testes e das figuras apresentadas é possível observar que o número de *checkpoints* mantidos foi de até 34 vezes menor que a versão anterior. Por outro lado houve um aumento de quase 1,5 vezes no número de mensagens enviadas, de mensagens

recebidas e do tempo total de execução do modelo. Desta forma, para esse modelo, a perda em processamento é muito pequena comparado com o ganho em espaço de armazenamento.

Também é possível perceber que os resultados dependem do modelo. Em testes não documentados neste trabalho, com um limite de 50 *checkpoints*, foi possível observar uma diminuição no tempo de simulação e também no número de mensagens enviadas e recebidas, pois os algoritmos foram chamados menos vezes durante a simulação. Entretanto houve um aumento de *checkpoints* mantidos, pois um limite maior permite o armazenamento de mais *checkpoints* até que seja chamado o método de exclusão.

Desta forma, cabe ao modelador escolher um limite onde as perdas em processamento sejam aceitáveis diante dos ganhos em espaço de armazenamento.

5 CONCLUSÃO

Este trabalho apresentou uma estratégia para limitar o espaço de memória utilizado por *checkpoints*. Um algoritmo foi proposto e implementado no DCB, onde foram feitos testes para validar a estratégia.

O algoritmo proposto diminuiu em até 34 vezes o número de *checkpoints* mantidos em comparação com a versão anterior do DCB, diminuindo assim a memória utilizada pela simulação. Desta forma simulações grandes podem ser executadas sem serem interrompidas por falta de memória.

Entretanto os experimentos também apresentaram um aumento de 1,5 vezes no número de mensagens trocadas e no tempo total da simulação, que em algumas simulações podem se tornar um problema. Assim, fica por responsabilidade do modelador escolher a melhor combinação entre tamanho do espaço de armazenamento e tempo de execução da simulação.

5.1 Trabalhos futuros

Trabalhos futuros podem tentar encontrar o melhor número para limitação de *checkpoints* armazenados, para que a troca de mensagens seja menor e consequentemente o tempo de simulação diminua.

A forma como são chamados os algoritmos de busca da *recovery line* e exclusão dos *checkpoints* também podem ser melhorados. Na versão atual estes são chamados quando o buffer de *checkpoints* alcança 70% e 90% de sua capacidade, respectivamente. Encontrar uma maneira melhor para as chamadas dos algoritmos também poderá impactar positivamente no tempo de execução da simulação.

Também fica como trabalho futuro a descoberta de até onde é válida a implementação do algoritmo proposto, ou seja, quando os prós trazidos pela limitação da memória superam os contras causados por essa limitação.

REFERÊNCIAS

CAO, G.; SINGHAL, M. On Coordinated Checkpointing in Distributed Systems. **IEEE Trans. Parallel Distrib. Syst.**, Piscataway, NJ, USA, v.9, n.12, p.1213–1225, Dec. 1998.

CAROTHERS, C. D.; BAUER, D.; PEARCE, S. ROSS: a high-performance, low memory, modular time warp system. In: FOURTEENTH WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2000. p.53–60. (PADS '00).

DAS, S. R.; FUJIMOTO, R. M. An Adaptive Memory Management Protocol for Time Warp Parallel Simulation. **SIGMETRICS Perform. Eval. Rev.**, New York, NY, USA, v.22, n.1, p.201–210, May 1994.

ELNOZAHY, E. N. M. et al. A Survey of Rollback-recovery Protocols in Message-passing Systems. **ACM Comput. Surv.**, New York, NY, USA, v.34, n.3, p.375–408, Sept. 2002.

FERSCHA, A.; TRIPATHI, S. K. **Parallel and Distributed Simulation of Discrete Event Systems**. College Park, MD, USA: [s.n.], 1994.

FUJIMOTO, R. M. Parallel and Distributed Simulation Systems. , [S.l.], 2000.

FUJIMOTO, R. M. Parallel and Distributed Simulation Systems. **Proceedings of the 2001 Winter Simulation Conference**, [S.l.], 2001.

KUMAR, S.; CHAUHAN, R. K.; KUMAR, P. Real Time Snapshot Collection Algorithm for Mobile Distributed Systems with Minimum Number of Checkpoints. , [S.l.], 2010.

LAW, A. M.; KELTON, W. D. **Simulation Modeling and Analysis**. 2nd.ed. [S.l.]: McGraw-Hill Higher Education, 1997.

LIU, Y.; CHEN, J. Garbage collection in uncoordinated checkpointing algorithms. **Journal of Computer Science and Technology**, [S.l.], v.14, p.242–249, 1999.

MELLO, B. A. d. Co-Simulação Distribuída de Sistemas Heterogêneos. **Tese (Doutorado em Ciência da Computação)**, [S.l.], 2005.

MITTRA, S. A Virtual Memory Management Scheme for Simulation Environment. In: IEEE INTERNATIONAL VERILOG HDL CONFERENCE, 4., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1995. p.114–. (IVC '95).

PARIZOTTO, R.; MELLO, B. A. d. Heuristic checkpointing on a parallel heterogeneous discrete event simulation system. , [S.l.], 2017.

PIDD, M. An Introduction to Computer Simulation. In: CONFERENCE ON WINTER SIMULATION, 26., San Diego, CA, USA. **Proceedings...** Society for Computer Simulation International, 1994. p.7–14. (WSC '94).

PREISS, B. R.; MACINTYRE, I. D.; LOUCKS, W. M. On the Trade-off between Time and Space in Optimistic Parallel Discrete-Event Simulation. , [S.l.], 1992.

QUAGLIA, F. Combining Periodic and Probabilistic Checkpointing in Optimistic Simulation. In: THIRTEENTH WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1999. p.109–116. (PADS '99).

SATO, K. et al. Design and Modeling of a Non-blocking Checkpointing System. In: INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING, NETWORKING, STORAGE AND ANALYSIS, Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 2012. p.19:1–19:10. (SC '12).

SHANNON, R. E. Introduction to the Art and Science of Simulation. In: CONFERENCE ON WINTER SIMULATION, 30., Los Alamitos, CA, USA. **Proceedings...** IEEE Computer Society Press, 1998. p.7–14. (WSC '98).

WANG, Y.-M. **Space Reclamation for Uncoordinated Checkpointing in Message-passing Systems**. 1993. Tese (Doutorado em Ciência da Computação) — , Champaign, IL, USA. UMI Order No. GAX94-11816.

YOUNG, C. H.; WILSEY, P. A. Optimistic Fossil Collection for Time Warp Simulation. **Proceedings of the 29th Annual Hawaii International Conference on System Sciences**, [S.l.], 1996.